

Compression and Sieve: Reducing Communication in Parallel Breadth First Search on Distributed Memory Systems

Huiwei Lv^{*†}, Guangming Tan^{*}, Mingyu Chen^{*}, Ninghui Sun^{*}

^{*}State Key Laboratory of Computer Architecture,
Institute of Computing Technology, Chinese Academy of Sciences

[†]Graduate University of Chinese Academy of Sciences
Beijing, China 100190

Email: lvhuiwei@ncic.ac.cn, tgm@ict.ac.cn, cmv@ict.ac.cn, snh@ncic.ac.cn

Abstract—For parallel breadth first search (BFS) algorithm on large-scale distributed memory systems, communication often costs significantly more than arithmetic and limits the scalability of the algorithm. In this paper we sufficiently reduce the communication cost in distributed BFS by compressing and sieving the messages. First, we leverage a bitmap compression algorithm to reduce the size of messages before communication. Second, we propose a novel distributed directory algorithm, cross directory, to sieve the redundant data in messages. Experiments on a 6,144-core SMP cluster show our algorithm outperforms the baseline implementation in Graph500 by 2.2 times, reduces its communication time by 79.0%, and achieves a performance rate of 12.1 GTEPS (billion edge visits per second).

I. INTRODUCTION

Recently, graph has been extensively used to abstract complex systems and interactions in emerging “big data” applications, such as social network analysis, World Wide Web, biological systems and data mining. With the increasing growth in these areas, petabyte-sized graph datasets are produced for knowledge discovery [1], [2], which could only be solved by distributed machines; benchmarks, algorithms and runtime systems for distributed graph have gained much popularity in both academia and industry [3], [4], [5], [6]. One of the most widely used graph-searching algorithms is breadth-first search (BFS), which serves as a building block for a great many graph algorithms such as minimum spanning tree, betweenness centrality, and shortest paths [7], [8], [9], [10].

Implementing a distributed BFS with high performance, however, is a challenging task because of its expensive communication cost [11], [2]. Generally, algorithms have two kinds of costs: arithmetic and communication. For distributed algorithms, communication often costs significantly more than arithmetic. For example, on a 512-node cluster, the baseline BFS algorithm in Graph 500 spends about 70% time on communication during its traversal on a scale-free graph with 8 billion vertices (Figure 1). Therefore the most critical task in a distributed BFS algorithm is to minimize its communication.

Several different approaches are proposed to optimize communication in distributed BFS (Table I): using two-dimensional partitioning of the graph to reduce communication

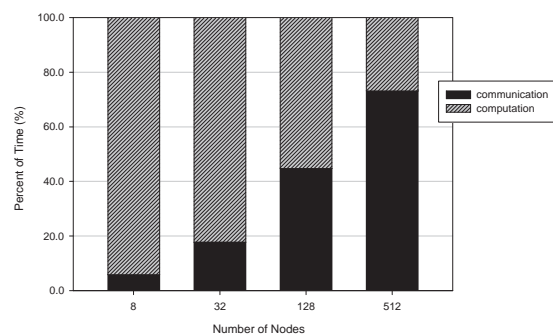


Fig. 1. Time breakdown of a baseline distributed BFS in a weak scaling experiment that use fixed problem size per node (each node has about 16M vertices).

TABLE I
COMPARISON OF VARIOUS APPROACHES FOR REDUCING
COMMUNICATION COST IN DISTRIBUTED BFS.

Approach	Category
Two-dimensional partitioning [4], [5]	algorithm
Bitmap & sparse vector [3], [5]	data structure
PGAS with communication coalescing [12]	runtime
<i>This Work</i> : compression & sieve	data structure

overhead [4], [5], using bitmap or sparse vector to reduce the size of messages [3], [5], or applying communication coalescing in PGAS implementation to minimize message overhead [12]. These approaches attack the problem from different angles: algorithm, data structure and runtime. In this paper, we will focus on reducing the size of communication messages (the optimization of data structures). The main techniques we use are *compression* and *sieve*. Overall, we make the following contributions:

- By compressing the messages, we reduce the communication time by 52.4% and improved its overall performance by $1.7\times$ compared to the baseline BFS algorithm.

$P_1: \{0,1\}$ $P_2: \{2,3\}$ $P_3: \{4,5\}$ $P_4: \{6,7\}$

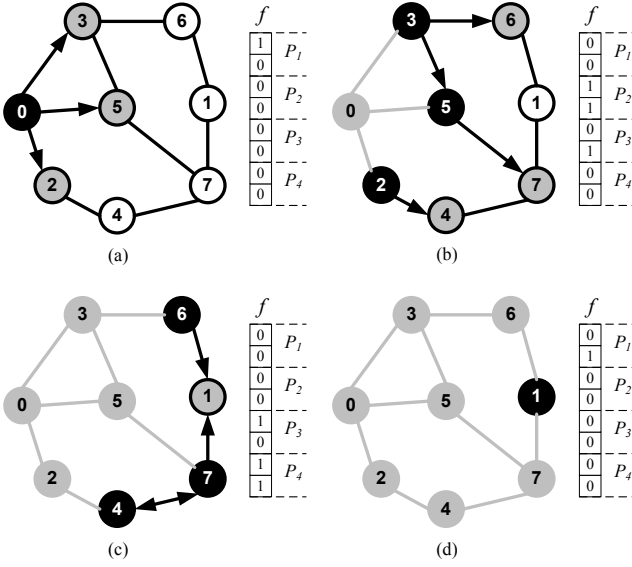


Fig. 2. The operation of BFS on an undirected graph. The frontier f is represented as a vector.

- By sieving the messages with a novel distributed directory before compression. We further reduce the communication by 55.9% and improved the performance by another 1.3 \times , achieving a total 79.0% reduction in communication and 2.2 \times performance improvement over the baseline implementation.
- We implement and analyse several compression methods for bitmap compression. Our experiment shows the space-time tradeoff of different compression methods.

In the next section we will introduce the problem with an example. Section III will describe the baseline BFS algorithm. Section IV and Section V will describe our BFS algorithms with compression and sieve. The analysis and experiment results are presented in Section VI and Section VII, followed by related works and concluding remarks in Section VIII and Section IX.

II. MOTIVATION

We start with an example illustrating the breadth-first search (BFS) algorithm. Given a graph $G = (V, E)$ and a distinguished source vertex s , breadth-first search systematically explores the edges of G to “discover” every vertex that is reachable from s . In Figure 2, the source vertex 0 is painted black when the algorithm begins. Then it explores its adjacent vertices: 3, 5 and 2, and paints them black. The exploration goes on until all vertices are visited. Vertices discovered the first time is painted black; discovered vertices are painted solid grey; vertices to be discovered are painted grey with black edge. The frontier f of the graph is the set of the vertices which are discovered the first time.

For distributed BFS, the vertices as well as the frontier are divided among processors: $P_1 : \{0,1\}$, $P_2 : \{2,3\}$, $P_3 : \{4,5\}$, $P_4 : \{6,7\}$. And the global information of the

TABLE II
THE SIZE OF THE FRONTIER, REPRESENTED AS BITMAP OR SPARSE VECTOR, AT EACH LEVEL OF BFS OF A SCALE-FREE GRAPH WITH 1.6 BILLION VERTICES. FOR SPARSE VECTOR, EACH VERTEX IS REPRESENTED AS A 64-BIT NUMBER.

Level	#Vertices	bitmap	sparse vector
1	2	196.9MB	16B
2	20842	196.9MB	162.8KB
3	235274348	196.9MB	2.0GB
4	1377666413	196.9MB	10.2GB
5	38582585	196.9MB	294.4MB
6	88639	196.9MB	692.4KB
7	211	196.9MB	1.69KB
Total	1651633040	1.4GB	12.4GB

frontier can only be retrieved through communication. For P_1 in this example, it only “owns” the information of whether vertex 0 and 1 are visited. If it want to identify whether vertex 2 is visited, it needs to ask this information from P_2 . The common way to update the global f is to use MPI collective communication like ALLGATHER at the end of each level [4], [5], [3].

The most critical task for distributed BFS is to reduce the size of the frontier, which directly influence the size of the messages communicated. To reduce it, bitmap or sparse vector is commonly used to represent the frontier. Bitmap use a vector of size $|V|$ to represent the frontier, each bit of the vector representing a vertex: 1 means it is included in the frontier, 0 means it is not. Sparse vector includes the frontier vertices only, each is represented using 64 bits. For graphs of diameter d , bitmap is generally better when $d < 64$. Table II provides an example of the size of the frontier represented as bitmap or sparse vector, for a scale-free graph of 1.6 billion vertices. In this case, for $d = 7$, the total size of messages using bitmap is 1.4 GB, much less than the sparse vector’s 12.4 GB. Despite the huge space saved by bitmap, there remains two problems:

- The problem of bitmap is that it need to contain *all* the vertices to keep the position information of each vertex. For the above example, to represent 2 vertices at level 1, the size of the bitmap frontier is still 196.9MB, where most of the elements are zero. Fortunately, these zeros can be condensed. *We leverage lossless compression to reduce the size of the bitmap.*
- The other problem is the expensive broadcast cost of the ALLGATHER collective communication, which broadcasts *all* vertices to *all* processors. In fact, each processor needs only a small fractions of the frontier. For example, in Figure 2 (b), P_2 does not need to send the information of vertex 2 to P_4 , because vertex 2 does not has a direct edge connecting to the vertices of P_4 . *We propose a distributed directory to sieve the bitmap vectors before compression, further reducing its message size.*

Algorithm 1: A baseline distributed BFS

Input : s : source vertex id

```

1  $f(s) \leftarrow s$ ;
2 foreach processor  $P_i$  in parallel do
3   while  $f \neq \emptyset$  do
4      $t_i \leftarrow A_i \otimes f$ ;
5      $t_i \leftarrow t_i \odot \overline{\pi_i}$ ;  $\pi_i \leftarrow \pi_i + t_i$ ;
6      $f_i \leftarrow t_i$ ;
7      $f \leftarrow \text{ALLGATHERV}(f_i, P_i)$ ;
```

III. BASELINE BFS WITH BITMAP AS FRONTIER

A. BFS Described in Linear Algebra

Let A denote the adjacency matrix of the graph G , f_{Lk} denote the frontier at level k , and $\pi_k = \bigcup_{i=1}^k f_{Li}$ denote the visited information of previous frontiers. The exploration of level k in BFS is algebraically equivalent to a sparse matrix vector multiplication (SpMV): $f_{L(k+1)} \leftarrow A^T \otimes f_{Lk} \odot \overline{\pi_k}$ (we will omit the transpose and assume that the input is pre-transposed for the rest of this section). For example, traversing from level one (Figure 2 (a)) to level two (Figure 2 (b)) is equivalent to the linear algebra below.

$$A^T \otimes f_{L0} \odot \overline{\pi_0} = \begin{pmatrix} 00110100 \\ 00000011 \\ 10001000 \\ 10000110 \\ 00100001 \\ 10010001 \\ 01010000 \\ 01001100 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \odot \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = f_{L1}$$

The syntax \otimes denotes the matrix-vector multiplication operation, \odot denotes element-wise multiplication, $(a_1, a_2, \dots, a_n)^T \odot (b_1, b_2, \dots, b_n)^T = (a_1 b_1, a_2 b_2, \dots, a_n b_n)^T$, and overline represents the complement operation. In other words, $\overline{v_i} = 0$ for $v_i \neq 0$ and $\overline{v_i} = 1$ for $v_i = 0$.

In Figure 2, BFS starts from vertex v_0 , thus $f_{L0} = \{v_0\}$, $f_{L1} = \{v_2, v_3, v_5\}$, $f_{L2} = \{v_4, v_6, v_7\}$, $f_{L3} = \{v_1\}$. If we use a vector of size n to represent the corresponding frontier f_{Lk} , for example, $f_{L2} = \{0, 0, 1, 1, 0, 1, 0, 0\}$. This algorithm becomes deterministic with the use of (select, max)-semiring, because the parent is always chose to be the vertex with the highest label.

B. Baseline BFS

Algorithm 1 describes the baseline BFS. Each loop block (starting in line 3) performs a single level traversal. f represents the current frontier, which is initialized as an empty bitmap; t is a bitmap that holds the temporary parent information for that iteration only; π is the visited information of previous frontiers. The computational step (line 4,5,6) can be efficiently parallelized with multithreading. For SpMV operation in line 4, the matrix data is naturally splitted into pieces

Algorithm 2: Distributed BFS with compression.

```

1  $f(s) \leftarrow s$ ;
2 foreach processor  $P_i$  in parallel do
3   while  $f \neq \emptyset$  do
4      $t_i \leftarrow A_i \otimes f$ ;
5      $t_i \leftarrow t_i \odot \overline{\pi_i}$ ;
6      $\pi_i \leftarrow \pi_i + t_i$ ;  $f_i \leftarrow t_i$ ;
7      $f'_i \leftarrow \text{Compress}(f_i)$ ;
8      $f' \leftarrow \text{ALLGATHERV}(f'_i, P_i)$ ;
9      $f \leftarrow \text{Uncompress}(f')$ ;
```

TABLE III
A WAH COMPRESSED BITMAP.

16 bits	1000000000000000
3-bit groups	100 000 000 000 000 0
WAH	0100 1100 0000

for multithreading. At the end of each loop, ALLGATHER updates f with MPI collective communication.

IV. BFS WITH COMPRESSION

For large graphs, the communication time of distributed BFS algorithms can take as much as seventy percent of the total execution time. To reduce it, we need to reduce the size of the messages. One simple way is to use lossless compression, trading computation for bandwidth.

Algorithm 2 describe the distributed BFS with compression. The difference between Algorithm 2 and Algorithm 1 are line 7 and 9. At line 7 the frontier vector f is first compressed into f' before communication. At line 9 f' is uncompressed back to f after communication.

We use word-aligned hybrid (WAH) [13] for *Compress* and *Uncompress* function, as WAH is fast and well suited for bitmap compression. Table III shows the WAH compressed representation of 16 bits. In WAH, there are three types of words: literal words, fill words and active words. The most significant bit of a word is used to distinguish between a literal word (0) and a fill word (1). And a active word stores the last few bits. We assume that each computer word contains 4 bits and all fill bits are 0 in this example. Under this assumption, each literal word stores 3 bits from the bitmap, and each fill word represents a multiple of 3 bits. The second line in Table III shows the bitmap as 3-bit groups. The last line shows the WAH words. The first two words are regular words, the first is a literal word, and the second a fill word. The fill word 1100 indicates a 0-fill of 4 words long (containing 12 consecutive 0 bits). Note that the fill word stores the fill length as 4 rather than 12. The third word is the active word; it stores the last few bits that could not be stored in a regular word. For sparse bitmaps, where most of the bits are 0, a WAH compressed bitmap would consist of pairs of a fill word and a literal word [13].

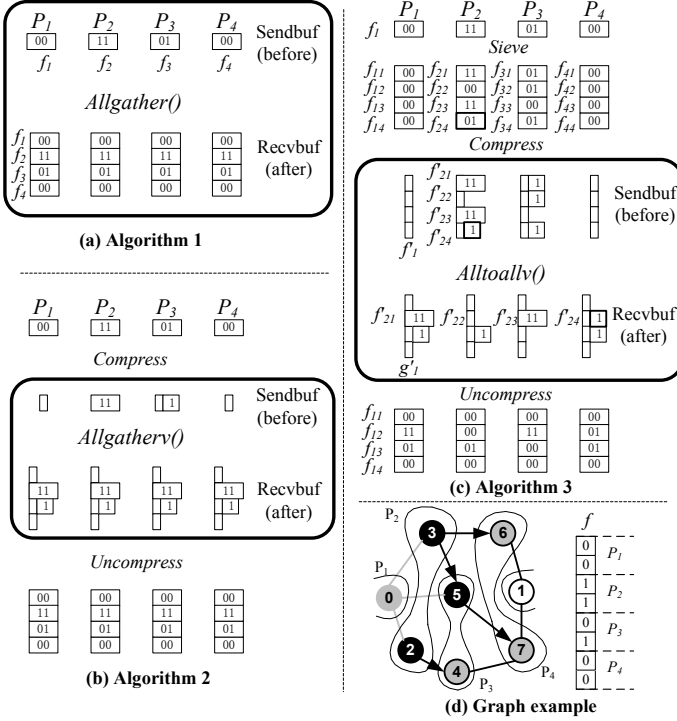


Fig. 3. Three different ways of communication.

Other lossless compression methods include run-length encoding, huffman coding, LZ77 [14], or more dedicated bitmap compression method such as byte-aligned bitmap compression (BBC) [15] and position list word aligned hybrid (PLWAH) [16]. There is a space-time tradeoff among these compression schemes. Comparing to WAH, LZ77 is slower but has a better compression ratio. The benefit of compression will depend on many factors such as compression ratio, sparsity of the messages, compression speed and network bandwidth. The best compression scheme can not be determined beforehand, so we use experiment to analyse these tradeoffs. Details will be presented in Section VII.

V. BFS WITH COMPRESSION AND SIEVE

The message size in the communication is reduced after compression. But there is still room for improvement. To achieve a better compression ratio, we can use a directory to sieve the bitmap, making it even sparser for compression.

In this section we propose a distributed directory, *cross directory*, as a sieve to reduce the number of messages sent to each processor. We will first introduce the data structure of cross directory in subsection V-A, then describe our BFS algorithm with compression and sieve in subsection V-B.

A. Cross Directory

The problem of collective communication like ALLGATHERV is that it sends all frontier vertices to all the processors — just like snoopy cache coherence algorithms, all updates are visible to all processors — regardless whether a vertex is meaningful to each processor. Take a look at Figure 3 (a),

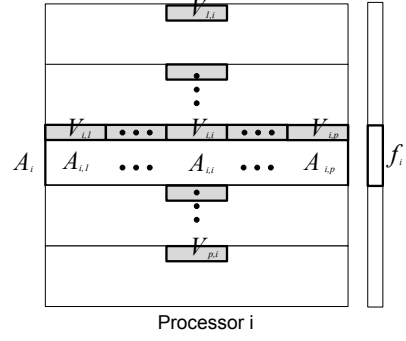


Fig. 4. The *cross directory* data structure for processor i .

after the ALLGATHER communication, each processor actually get all the frontier vectors. In fact, *each processor needs only a small fraction of the frontier*, and this fraction can be determined before communication. For example, in Figure 3 (d), P_4 only needs v_3 from P_2 . This means P_2 does not need to send the information of v_2 to P_4 , because v_2 does not has a direct edge connecting to the vertices of P_4 .

To explain this in algebra, we first partition the matrix A into p block-rows. Then partition each block A_i into p sub-blocks.

$$A \otimes f = \begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_p \end{pmatrix} \quad (1)$$

$$A_i = (A_{i,1} \ A_{i,2} \ \cdots \ A_{i,p}) \quad (2)$$

To calculate $f_4 = \sum_{i=1}^4 A_{4,i} \otimes f_i$,

$$A_{4,2} \otimes f_2 = \begin{pmatrix} 01 \\ 00 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}, \quad (3)$$

because $a_{0,0}$ and $a_{1,0}$ of $A_{4,2}$ are always zero (denote $A_{i,j} = [a_{i,j}]_{m \times n}$), y_1 will always be zero. So P_2 does not need to send x_1 to P_4 . We define a data structure to record this information and use it to sieve communication messages.

We formally define *directory vector* as follows: for each item v_k in vector $V_{i,j}$, v_k is set to one if column k in $A_{i,j}$ contains at least one non-zero.

$$V_{i,j} = (v_1, v_2, \dots, v_n) \quad (4)$$

$$\text{where } v_k = \begin{cases} 1, & \exists a_{i,k} = 1, i \in [1, m], k \in [1, n] \\ 0, & \text{otherwise} \end{cases}$$

For the above example, $V_{4,2} = (0, 1)$ is sent to P_2 from P_4 during initialization. When traversing begins, f_2 is sieved into $f_{2,4} = f_2 \odot V_{4,2} = (1, 1)^T \odot (0, 1)^T = (0, 1)^T$, so we only send one vertex (in compressed bitmap format) back instead of two. This “sieve effect” is where communication is reduced.

And the *cross directory* of processor P_i is defined as:

$$\mathbb{C}_i = \{V_{x,i} \text{ or } V_{i,x} \mid x = 1, 2, \dots, p\} \quad (5)$$

Besides a row of directory vectors $V_i = \{V_{i,y} \mid y = 1, 2, \dots, p\}$, P_i own a copy of the directory vectors $\{V_{x,i} \mid$

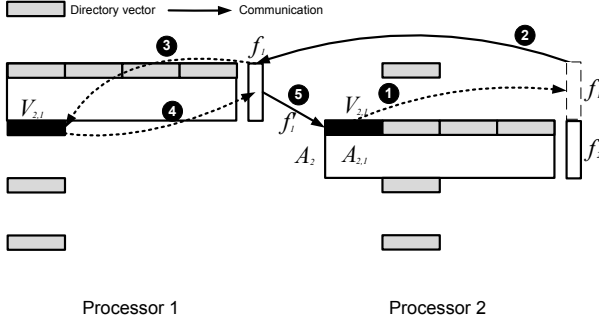


Fig. 5. Communications in Directory-based BFS algorithm. Example of multiply $A_{2,1}$ with f_1 in five steps.

Algorithm 3: Distributed BFS with sieving and compression.

Data: $f'_i = \{f'_{i,1}, f'_{i,2}, \dots, f'_{i,n-1}\}$:send buffer;
 $g'_i = \{g'_{i,1}, g'_{i,2}, \dots, g'_{i,n-1}\}$:receive buffer;
 \mathbb{C}_i :cross directory for P_i .

```

1  $f(s) \leftarrow s$ ;
2 initialize  $\mathbb{C}_i$ ;
3 foreach processor  $P_i$  in parallel do
4   while  $f \neq \emptyset$  do
5      $t_i \leftarrow \sum_{j=1}^n A_{i,j} \otimes f_{i,j}$ ;
6      $t_i \leftarrow t_i \odot \overline{\pi}_i$ ;
7      $\pi_i \leftarrow \pi_i + t_i$ ;  $f_i \leftarrow t_i$ ;
8     foreach  $j \in [0, n)$  in parallel do
9        $f_{i,j} = f_i \odot V_{j,i}$ ; /* sieving */;
10       $f'_{i,j} \leftarrow \text{Compress}(f_{i,j})$ ;
11       $g'_i \leftarrow \text{ALLTOALLV}(f'_i, P_i)$ ;
12      foreach  $j \in [0, n)$  in parallel do
13         $f_{i,j} \leftarrow \text{Uncompress}(g'_{i,j})$ ;

```

$x = 1, 2, \dots, p\}$ in column i . The directory in the column direction is established during initialization and used to provide a local lookup for sieving (See Figure 4).

Figure 5 illustrates an example of communication with cross directory. The matrix is row-block partitioned among four processors. $A_{2,1} \otimes f_1$ is done in five steps, $A_{2,1}$ need to get f_1 (step 1), P_2 then send a request message to P_1 (step 2), P_1 check its local copy of $V_{2,1}$ (step 3) and sieve f_1 with the non-zero positions (step 4), then P_1 send back a sieved f'_1 (step 5). The sieved vector is very sparse and can be represented as sparse vector, reducing the communication cost.

B. Sieve with Cross Directory

Algorithm 3 is our directory-based algorithm with compression and sieve: based on Algorithm 1, Algorithm 3 first sieves the frontier bitmap with the cross directory (line 9), making it sparser; then it compresses this sieved bitmap (line 10) and send it with ALLTOALLV (line 11); after received the compressed bitmap, the original vector could be restored with uncompression (line 13).

This *cross directory* is inspired by Pinar and Hendrickson's distributed directory [17] and Baker et al.'s assumed partition algorithm [18]. In their work, the communication pattern is dynamically determined and more general, while in our case, the communication parties are static. So we store the directory on both side of the communication, and update them synchronously on each side instead of send the updated directory over the network each time. Another difference lies in the collective communication. In Baker et al.'s assumed partition algorithm, point-to-point rendezvous communication is used, we find that could be replaced with a more efficient ALLTOALLV. More generally, the cross directory is applicable to matrix-vector multiplication when following premises are true: 1) the partition of the matrix is static so that communication parties are static; 2) the matrix remains unchanged and multiplication takes many times so that cross directory could be reused and its initialization cost could be omitted. For example, sum of the multiplication of the same matrix with different vectors ($\sum_{i=1}^n A x_i$).

C. Proof of Correctness

In this subsection we prove the correctness of Algorithm 3 by proving its equivalence to Algorithm 1.

Lemma 5.1: $A_{i,j} \otimes f_j = A_{i,j} \otimes f_j \odot V_{j,i}$.

Proof: Let $X = (x_1, x_2, \dots, x_n)^T = A_{i,j} \otimes f_j$, $Y = (y_1, y_2, \dots, y_n)^T = A_{i,j} \otimes f_j \odot V_{j,i} = (x_1 v_1, x_2 v_2, \dots, x_n v_n)^T$, $V_{j,i} = (v_1, v_2, \dots, v_n)^T$, and $f_j = (z_1, z_2, \dots, z_n)^T$. Denote $A_{i,j} = [a_{i,j}]_{m \times n}$, then $x_k = \sum_{l=1}^n a_{k,l} z_l$. According to the definition of directory vector, if $v_k = 0 \Rightarrow \forall a_{k,i} = 0, i \in [1, n] \Rightarrow x_k = \sum_{l=1}^n a_{k,l} z_l = 0$, so $y_k = x_k v_k = 0 = x_k$; if $v_k = 1 \Rightarrow x_k = x_k v_k = y_k$. Thus $X = Y$. ■

Lemma 5.2: $t_i = \sum_{j=1}^n A_{i,j} \otimes f_{i,j}$ in Algorithm 3 (line 5) is equivalent to $t_i = A_i \otimes f$ in Algorithm 1 (line 4).

Proof: In Algorithm 3, for $\forall j \in [1, n]$, $f_{i,j} = f_i \odot V_{j,i}$, according to Lemma 5.1, $\sum_{j=1}^n A_{i,j} \otimes f_{i,j} = \sum_{j=1}^n A_{i,j} \otimes f_j \odot V_{j,i} = \sum_{j=1}^n A_{i,j} \otimes f_j = A_i \otimes f = t_i$. ■

VI. ALGORITHM ANALYSIS

In this subsection we'd like to analyse the communication and space cost of the three algorithms in this paper.

A. Communication Cost

We study the parallel BFS problem in the message passing model of distributed computing: every processor has its own local memory, and data exchange between processors are done by message passing. The time taken to send a message between any two processors can be modeled as $T(n) = \alpha + n\beta$, where α is the latency (or startup time) per message, independent of message size, β is the transfer time per byte (inverse of bandwidth), and n is the number of bytes transferred [19]. This time cost model is generally used to model data movement either between levels of a memory hierarchy or over a network connecting processors. In this paper, we focus on the latter case. To simplify the analysis, we assume bandwidth cost is much bigger than latency cost ($n\beta \gg \alpha$), — as the dataset of

distributed BFS is big, — therefore $T(n)$ will be dominated by the bandwidth cost $n\beta$. For a given network, β is constant, so the communication cost is in direct proportion to the message size n . Let *communication volume of a processor* \mathcal{V}_i be the size of all messages communicated on processor P_i in an algorithm. The *communication volume of an algorithm* is defined as $\mathcal{V} = \max\{\mathcal{V}_i \mid i \in [1, p]\}$.

The communication volume of MPI collective communication is derived from [20], [21]: For p processors, when each processor needs to broadcast n/p size of message to others, the communication volume of both allgather and alltoall are $O(n)$. There are many algorithms for allgather, for example, ring and recursive doubling [20]. The time taken for these two algorithm is $T_{ring} = (p-1)\alpha + \frac{p-1}{p}n\beta$ and $T_{rec_dbl} = \log p\alpha + \frac{p-1}{p}n\beta$, respectively. No matter what algorithm is used, the bandwidth cost is the same $\frac{p-1}{p}n\beta$. In data-intensive applications like BFS, we assume bandwidth cost is much bigger than latency cost, so its communication volume is bound to $O(n)$. The communication volume of alltoall can be done in the same manner [21].

For graph $G(V, E)$, let $m = |E|$, $n = |V|$, let d be the diameter of the graph. At each level of BFS, the communication volume of allgather (Algorithm 1, line 7) is $O(n)$; the algorithm will finish at level d . So the communication volume of Algorithm 1 is $d \times O(n)$.

For Algorithm 2, let $C_i (C_i > 1)$ be the compression ratio of the *Compression* function of Algorithm 2 (line 7) at level i , let $C = \frac{1}{d} \sum_{i=1}^d \frac{1}{C_i} (C < 1)$ be the compression ratio factor. The communication volume of Algorithm 2 is $Cd \times O(n)$.

For Algorithm 3, let p be the number of the processors, $e = m/n$ be the average degree of a vertex, and C' be the compression ratio factor of Algorithm 3. The communication volume of Algorithm 3 is $C'd \times O(n)$. After sieve, a vertex is sent to at most $\min(e, p)$ processors in Algorithm 3 instead of p in Algorithm 1 and 2. Thus Algorithm 3's messages will contain less nonzeros than Algorithm 2's, which leads to a higher compression ratio and a smaller $C' (C' < C)$.

B. Memory Consumption

For Algorithm 1, the memory consumption of f is $O(n)$; t_i and π_i are $O(n/p)$. So the memory consumption of each processor of Algorithm 1 is $O(n)$.

Compared to Algorithm 1, Algorithm 2 replace f with f' , the memory consumption of which is at most as that of f , $O(n)$. So the memory consumption of each processor of Algorithm 2 is also $O(n)$.

Compared to Algorithm 2, Algorithm 3 added V_i , which costs $O(n)$ memory. So the memory consumption of Algorithm 3 is also bound to $O(n)$.

VII. EXPERIMENTAL RESULTS

This section presents experimental results for the distributed BFS.

TABLE IV
EXPERIMENT PLATFORM

System	SMP Cluster
Number of Nodes	512
Number of CPUs / node	2
Processor	Intel X5650
Number of cores	6
Number of threads	12
Core frequency	2.66 GHz
L1 cache size	384 KB
L2 cache size	1536 KB
L3 cache size	12 MB
Memory type	DDR3-1333
QPI Speed	6.4 GT/s
Interconnect	Infiniband
Rate	40 Gb/sec (4X QDR)

A. Experiment Setup

Our performance results is collected on a 512-node multi-core cluster system, connected by Infiniband of 40 Gb/s. Each node has an SMP architecture with two Xeon X5650 CPUs (Westmere), which are connected through Intel QuickPath Interconnect (QPI) of 6.4 GT/s. The Xeon X5650 has six cores, each supports simultaneous multithreading (SMT) up to two threads. Each node has 24GB DDR3-1333 RAM. In our experiments we used up to 512 node, or 6,144 cores, to run the experiment. We use gcc 4.3.4 and MPICH2 1.4.1 to compile our algorithms. The GNU OpenMP library is used for intra-node threading. See Table IV.

Our algorithms are based on Graph 500 benchmark. Input datasets are generated use synthetic kronecker graphs [22] which follow power law distributions: heavy tails for the degree distribution; small diameters; and densification and shrinking diameters over time. That means most of vertices has a small number of neighboring vertices and the graph is sparse. The graph size is determined by two parameters: “Scale” and “Edge factor”, where the total number of vertices N equals 2^{Scale} , and the number of edges, $M = edgefactor * N$. The default edgefactor is set 16. In order to save space, an adjacent array (or list) representing sparse graph is transformed into compressed sparse row (CSR) or column (CSC). We focus on the CSR-based BFS implementation in Graph 500. In order to compare the performance of Graph 500 implementations across a variety of architectures, a new performance metric is adopted in Graph 500. Let *time* be the measured execution time for running BFS. Let m be the number of input edge tuples within the component traversed by the search, counting any multiple edges and self-loops. The normalized performance rate *traversed edges per second (TEPS)* is defined as: $TEPS = m/time$.

Table V lists different BFS algorithms tested in our experiment.

TABLE V
DIFFERENT BFS ALGORITHMS TESTED

Name	Algorithm Details
<i>BIT</i>	Baseline BFS with bitmap (Algorithm 1)
<i>WAH</i>	BFS with WAH compression (Algorithm 2)
<i>DIR-WAH</i>	Directory-based BFS with WAH compression (Algorithm 3)

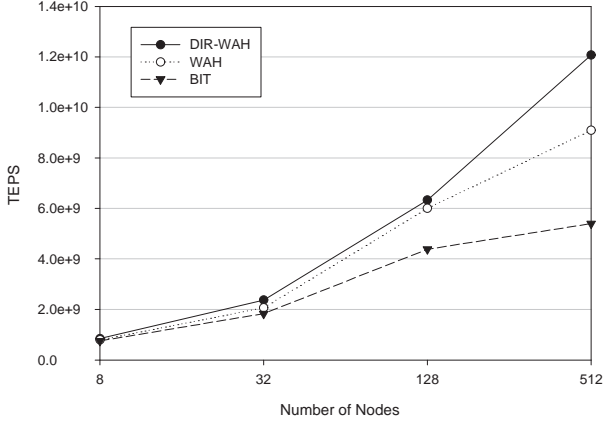


Fig. 6. Weak scaling performance of different BFS algorithms. The experiment use fixed problem size per node (each node has about 16M vertices).

B. Experiment Results

Figure 6 shows the weak scaling performance of our BFS algorithms. We run this experiment on our 512-node SMP cluster, with one process per SMP node. For intra-node threading, we use the GNU OpenMP library. Algorithm 3 (*DIR-WAH*) outperforms all other algorithms and have the best scalability. *DIR-WAH* achieves $1.21\text{E}+10$ TEPS at scale 33 with 512 nodes, $1.33\times$ than Algorithm 2 (*WAH*), and $2.24\times$ faster than Algorithm 1 (*BIT*). We can see the benefits of compression and sieve here: with compression, *WAH* is $1.69\times$ faster than *BIT*; with sieve, *DIR-WAH* is another $1.33\times$ than *WAH*. The performance gap between *DIR-WAH* and *BIT* becomes wider as the number of nodes increases. This is because the larger the number of nodes is, the more distributed BFS algorithm will depend communication, and the more benefits compression and sieve will bring. We will see the time breakdown in the next figure.

Figure 7 is the time breakdown of the algorithms in Figure 6: “traversing” time is the time spent on local computing; “reducing” time is the time spent on a MPI reduction operation to get the total vertex count of the frontier; “communication” time is the time spent on communication; “compression & sieve” time is the time spent on compression and sieve. For all three algorithms, as the number of nodes increases, “communication” times increase exponentially. For *BIT*, it accounts for as much as 73.2% of the total time for 512 node. The “reducing” times also increases because the imbalance of a graph become more severe as the graph becomes larger;

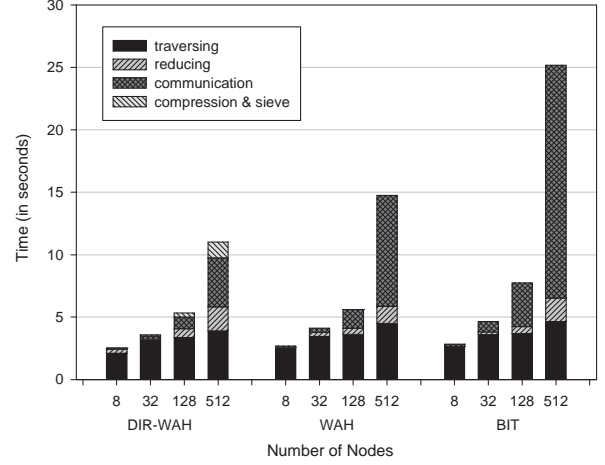


Fig. 7. Time breakdown of different BFS algorithms.

TABLE VI
DESCRIPTION OF PROFILED TIME IN FIGURE 7.

Label	Description
traversing	Local sparse matrix vector multiplication
reducing	MPI reduction to get vertex sum of current frontier
communication	Time spent on communication
compression & sieve	Time spent on compression and sieve

the local “traversing” times remain more or less the same because the problem size per node is fixed. At 512 node, *WAH* reduces the “communication” time by 52.4% compared to *BIT*; *DIR-WAH* reduces the “communication” time by another 55.9% compared to *WAH*, achieving a total 79.0% reduction compared to *BIT*, from 18.6 seconds to 3.9 seconds. On one hand, the “compression & sieve” time of *WAH* (only compression time is counted for *WAH*) at 512 nodes is less than 0.1% of the total run time and not shown in the figure. This means the benefit of compression is at very little cost. On the other hand, the time of “compression & sieve” in *DIR-WAH*, — the computing time traded for bandwidth — accounts for 11.1% of the total. This is because Algorithm 3 (line 9) needs to copy the frontier for each process before sieve. This copying time is expensive because it is in direct proportion to the number of processes. Overall, comparing *DIR-WAH* to *WAH* (512 nodes), sieve costs about 1.3 seconds but saves 5.0 seconds in communication — the saving is worth the cost.

Figure 8 plots the performance of different BFS algorithms at different scales. The experiment runs on 512 nodes. We can learn from this plot that the compression and sieve method favours larger messages. The size of messages will affect the results: at scale 26, *DIR-WAH*, *WAH* and *BIT* need to exchange 8MB bitmap globally using MPI collective communications; at scale 33, 1GB. *DIR-WAH* is the slowest when the scale

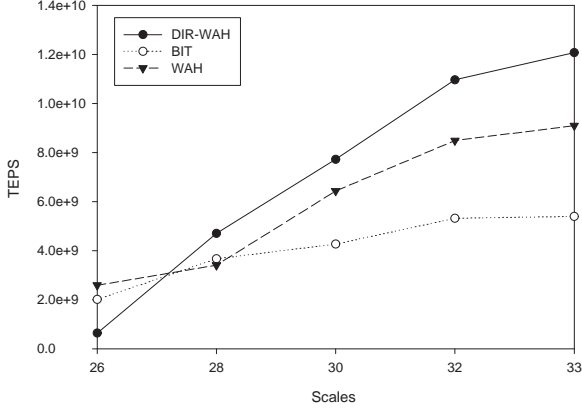


Fig. 8. Performance of different BFS algorithms at different scales. The experiment runs on 512 nodes.

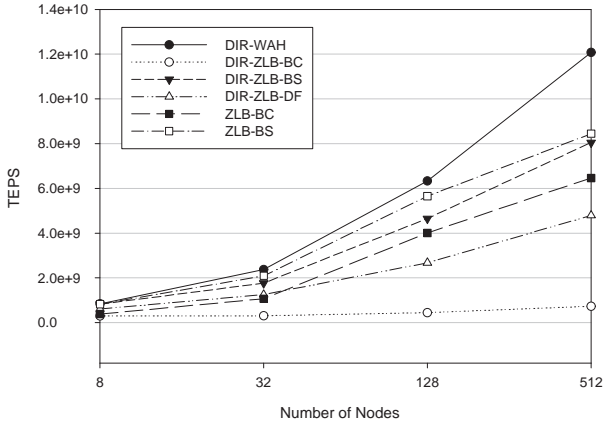


Fig. 9. Weak scaling performance result of BFS of different compression methods. The experiment use fixed problem size per node (each node has about 16M vertices).

is small, but it gradually catches up and surpasses all other algorithms when scale gets bigger.

As mentioned in section IV, different methods could be used for compression. We did not implement all of them but choose two, Zlib library [23] and WAH, based on following reasons: Zlib library is famous for good compression on a wide variety of data and provides different compression levels; WAH is dedicated to bitmap compression, simpler than PLWAH and faster than BBC. We use Zlib 1.2.6, and three different compression levels: best compression (*ZLB-BC*), best speed (*ZLB-BS*) and default (*ZLB-DF*). The results are plotted in Figure 9 and Figure 10.

Figure 9 shows the weak scaling performance of BFS algorithms with different compression and sieve methods. BFS with Zlib best compression *ZLB-BC* is the slowest. With 512 nodes, *DIR-WAH* provides the best performance, followed by *ZLB-BS* (69.9% of *DIR-WAH*), *DIR-ZLB-BS* (66.7%), *ZLB-BC* (53.5%), and *DIR-ZLB-DF* (39.7%) respectively.

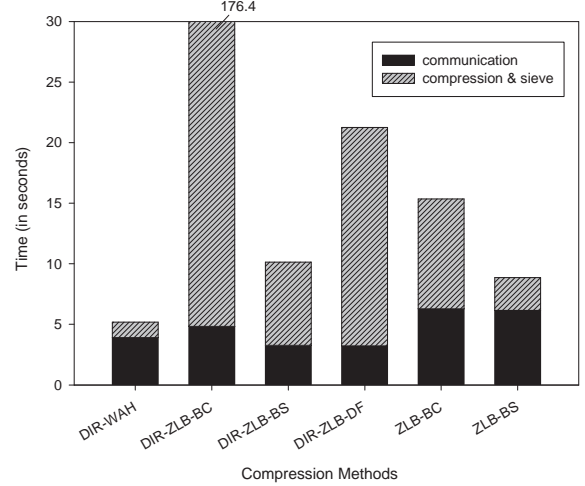


Fig. 10. Time profiling of different compression implementations.

Figure 10 shows the time breakdown of these algorithms. At scale 33 with 512 nodes, *DIR-ZLB-DF*'s "communication" time is the smallest, 0.82 \times of *DIR-WAH*, followed by *DIR-ZLB-BS* (0.83 \times), *DIR-ZLB-BC* (1.23 \times), *ZLB-BS* (1.57 \times) and *ZLB-BC* (1.61 \times). Although *DIR-ZLB-DF* and *DIR-ZLB-BS*'s communication times are less than *DIR-WAH*, their "compression and sieve" times are 14.25 \times and 5.44 \times of *DIR-WAH*. So the overall performance of *DIR-ZLB-DF* and *DIR-ZLB-BS* are worse than *DIR-WAH*. For all three compression levels in Zlib we tested, default method, not the best compression method, provides the best compression ratio. In fact, the Zlib best compression method is not suited for bitmap compression: it is not only the slowest, but also provides the worst compression ratio.

VIII. RELATED WORKS

Several different approaches are proposed to reduce the communication in distributed BFS. Yoo et al. [4] run distributed BFS on IBM BlueGene/L with 32,768 nodes. Its high scalability is achieved through a set of memory and communication optimizations, including a two-dimensional partitioning of the graph to reduce communication overhead. Buluç and Madduri [5] improved Yoo et al.'s work by adding hybrid MPI/OpenMP programming to optimize computation on state-of-the-art multicore processors, and managed to run distributed BFS on a 40,000-core machine. The method of two-dimensional partitioning reduces the number of processes involved in collective communications. Our algorithm reduces the communication overhead in a different way: minimizing the size of messages with compression and sieve. Moreover, these two optimizations could be combined together to further reduce the communication cost in distributed BFS. A preliminary result is presented in Section IX to demonstrate its potential. Beamer et al. [24] use a hybrid top-down and bottom-up approach that dramatically reduces the number of edges examined. The sample code in Graph 500 [3] use bitmap

(bitset array) in communication, reducing its message size. Cong et al. [12] applying communication coalescing in PGAS implementation to minimize message overhead.

Benchmarks, algorithms and runtime systems for graph algorithms have gained much popularity in both academia and industry. Earlier works on Cray XMT/MTA [25], [26] and IBM Cyclops-64 [8] prove that both massive threads and fine-grained data synchronization improve BFS performance. Bader and Madduri [25] designed a fine-grained parallel BFS which utilizes the support for hardware threading and synchronization provided by MTA-2, and ensures that the graph traversal is load-balanced to run on thousands of hardware threads. Mizell and Maschhoff [26] discussed an improvement on Cray XMT. Using massive number of threads to hide latency has long been employed in these specialized multi-threaded machines. With the recent progress of multi-core and SMT, this technique can be popularized to more commodity users. Both core-level parallelism and memory-level parallelism are exploited by Agarwal et al. [27] for optimized parallel BFS on Intel Nehalem EP and EX processors. They achieved performances comparable to special purpose hardware like Cray XMT and Cray MTA-2 and first identified the capability of commodity multi-core systems for parallel BFS algorithms. Scarpazza et al. [28] use an asynchronous algorithm to optimize communication between SPE and SPU for running BFS on STI CELL processors. Leiserson and Schardl [29] use Cilk++ runtime model to implement parallel BFS. Cong et al. [12] present a fast PGAS implementation of distributed graph algorithms. Another trend is to use GPU for parallel BFS, for they provide massively parallel hardware threads, and are more cost-effective than the specialized hardware. Generally, GPUs are good at regular problems with contiguous memory accesses. The challenge of designing an effective BFS algorithm on GPU is to solve the imbalance between threads and to hide the cost of data transfer between CPU and GPU. There are several works [30], [31], [32] working on this direction.

IX. CONCLUSION

The main purpose of this paper is to reduce the communication cost in distributed breadth-first search (BFS), which is the bottleneck of the algorithm. We found two problems in previous distributed BFS algorithms: first, their message formats are not condensed enough; second, broadcasting messages causes waste. We propose to reduce the message size by compressing and sieving. By compressing the messages, we reduce the communication time by 52.4%. By sieving the messages with a distributed directory before compression, we reduce the communication time by another 55.9%, achieving a total 79.0% reduction in communication time and 2.2 \times performance improvement over the baseline implementation.

For future works, we would like to combine our optimization of message size with other methods such as two-dimensional partitioning [5] and hybrid top-down and bottom-up algorithm [24]. The potential is clear. A preliminary optimization of the distributed BFS algorithm in combinational

BLAS library [33], compressing the sparse vector using Zlib library, reduces the communication time by 41.9% and increases overall performance by 1.11 \times . By using compressed bitmap and adding sieve, we expect to further improve its performance.

REFERENCES

- [1] D. A. Bader, "Petascale computing for large-scale graph problems," in *Proceedings of the 7th international conference on Parallel processing and applied mathematics*, ser. PPAM'07. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 166–169.
- [2] A. Lumsdaine, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 1, p. 5, 2007.
- [3] "The Graph 500 List," 2011. [Online]. Available: <http://www.graph500.org/>
- [4] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A scalable distributed parallel breadth-first search algorithm on bluegene/l," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, ser. SC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 25–.
- [5] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 65:1–65:12.
- [6] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing - "abstract"," in *Proceedings of the 28th ACM symposium on Principles of distributed computing*, ser. PODC '09. New York, NY, USA: ACM, 2009, pp. 6–6.
- [7] B. Chazelle, "A minimum spanning tree algorithm with inverse-ackermann type complexity," *J. ACM*, vol. 47, pp. 1028–1047, November 2000.
- [8] G. Tan, V. Sreedhar, and G. Gao, "Analysis and performance results of computing betweenness centrality on ibm cyclops64," *The Journal of Supercomputing*, vol. 56, pp. 1–24, 2011.
- [9] U. Brandes, "A faster algorithm for betweenness centrality*," *The Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.
- [10] B. Cherkassky, A. Goldberg, and T. Radzik, "Shortest paths algorithms: Theory and experimental evaluation," *Mathematical Programming*, vol. 73, pp. 129–174, 1996.
- [11] A. Chan, F. Dehne, and R. Taylor, "Cgmgraph/cgmlib: Implementing and testing cgm graph algorithms on pc clusters and shared memory machines," in *International Journal of High Performance Computing Applications*. Springer, 2005.
- [12] G. Cong, G. Almasi, and V. Saraswat, "Fast pgas implementation of distributed graph algorithms," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.
- [13] K. Wu, E. J. Otoo, and A. Shoshani, "Optimizing bitmap indices with efficient compression," *ACM Trans. Database Syst.*, vol. 31, no. 1, pp. 1–38, Mar. 2006.
- [14] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *Information Theory, IEEE Transactions on*, vol. 23, no. 3, pp. 337 – 343, may 1977.
- [15] G. Antoshenkov, "Byte-aligned bitmap compression," in *Proceedings of the Conference on Data Compression*, ser. DCC '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 476–.
- [16] F. Deligè and T. B. Pedersen, "Position list word aligned hybrid: optimizing space and performance for compressed bitmaps," in *Proceedings of the 13th International Conference on Extending Database Technology*, ser. EDBT '10. New York, NY, USA: ACM, 2010, pp. 228–239.
- [17] A. Pinar and B. Hendrickson, "Communication support for adaptive computation," 2001.
- [18] A. H. Baker, R. D. Falgout, and U. M. Yang, "An assumed partition algorithm for determining processor inter-communication," *Parallel Comput.*, vol. 32, pp. 394–414, June 2006.
- [19] V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

- [20] R. Thakur, "Improving the performance of collective operations in MPICH," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface. Number 2840 in LNCS, Springer Verlag (2003) 10th European PVM/MPI User's Group Meeting*. Springer Verlag, 2003, pp. 257–267.
- [21] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of mpi collective operations," *Cluster Computing*, vol. 10, pp. 127–143, June 2007.
- [22] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos, "Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication," in *Knowledge Discovery in Databases: PKDD 2005*, ser. Lecture Notes in Computer Science, A. Jorge, L. Torgo, P. Brazdil, R. Camacho, and J. Gama, Eds. Springer Berlin / Heidelberg, 2005, vol. 3721, pp. 133–145.
- [23] "Zlib Home Page," 2012. [Online]. Available: <http://zlib.net>
- [24] S. Beamer, K. Asanovi, and D. A. Patterson, "Searching for a parent instead of fighting over children: A fast breadth-first search implementation for graph500," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-117, Nov 2011. [Online]. Available: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-117.html>
- [25] D. A. Bader and K. Madduri, "Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2," in *Proceedings of the 2006 International Conference on Parallel Processing*, ser. ICPP '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 523–530.
- [26] D. Mizell and K. Maschhoff, "Early experiences with large-scale cray xmt systems," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–9.
- [27] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.
- [28] D. P. Scarpazza, O. Villa, and F. Petrini, "Efficient breadth-first search on the cell/be processor," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, pp. 1381–1395, October 2008.
- [29] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '10. New York, NY, USA: ACM, 2010, pp. 303–314.
- [30] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPOPP '11. New York, NY, USA: ACM, 2011, pp. 267–276.
- [31] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *Proceedings of the 14th international conference on High performance computing*, ser. HiPC'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 197–208.
- [32] L. Luo, M. Wong, and W.-m. Hwu, "An effective gpu implementation of breadth-first search," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 52–55.
- [33] A. Buluç and J. R. Gilbert, "The combinatorial blas: Design, implementation, and applications," in *International Journal of High Performance Computing Applications (IJHPCA)*, 2011.